THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 62, No. 1, January 1983
Printed in U.S.A.

The 3B20D Processor & DMERT Operating System:

DMERT Operating System

By M. E. GRZELAKOWSKI, J. H. CAMPBELL, and M. R. DUBMAN

(Manuscript received March 17, 1982)

General operating system services in Duplex Multiple Environment Real Time (DMERT) are provided by a kernel and a set of cooperating processes. These services support a multitude of process-oriented features that include a rich set of interprocess communication mechanisms and a sophisticated collection of memory manipulation primitives. The operating system provides processes with two scheduling alternatives and various creation and termination options. The cooperating processes include input/output drivers and device handlers that enable processes to communicate with a variety of peripheral devices. Another set of processes simulates a UNIX™ *operating system and file system.*

I. INTRODUCTION

This article describes the design of the DMERT operating system nucleus. The reader should be familiar with the basic architecture of Duplex Multiple Environment Real Time (DMERT), which is described in a companion paper.[1] The operating system nucleus is composed of: the kernel, which supports interprocess communication mechanisms, the system clock, and interrupts; the special processes, which perform memory management and scheduling; two supervisor processes, which handle process management and the *UNIX** operating system environment; and three kernel processes that control communication with peripherals and the file system.

In general, DMERT applications view the operating system nucleus

* Trademark of Bell Laboratories.

as one entity rather than a set of cooperating processes. Thus, the interrelationships between these processes are hidden from them. This article describes the design of DMERT's nucleus from the user's perspective; that is, it is feature oriented rather than process oriented. After finishing this paper, however, the reader should be familiar with both the feature set and the internal design of the cooperating processes.

In order for a process to use any of the operating system's features, it must interface to the operating system with Operating System Traps (OSTs) or Interprocess Communication (IPC) mechanisms. OSTs were discussed in Ref. 1, and Section II describes IPCs. Sections III, IV, and V define the image and life cycle of processes by describing how memory is handled, the scheduling options, and how processes are created and terminated. Section VI summarizes the characteristics of a special type of process, namely a user process. Section VII discusses the input/output (I/O) and disk interfaces, and Section VIII describes DMERT's file systems.

II. INTERPROCESS COMMUNICATION

As stated in the previous paper, DMERT provides an extensive set of interprocess communication and synchronization mechanisms, including messages, events, process ports, faults, interprocess traps, first in-first outs (FIFOs), and shared memory. These interprocess communication mechanisms are described below.

2.1 Events

An event is a one-bit piece of information that may be sent, received, and recognized by a process. DMERT has 32 event bits, several of which are reserved for special meanings by DMERT. Those not reserved may be used in any manner agreed upon by a set of cooperating processes. DMERT provides several event-type OSTs that enable one process to send an event to another process or to a class of processes.

Processes have multiple entry points for start, event, OST, and fault entries. The purpose of the event entry is to allow the system to direct the process's attention to an event that has just occurred. Thus, when any of the process's event bits are set, control is passed to its event entry.

Additional control over events is given to a kernel process. It has the ability to mask and reenable the bits that comprise its event flag word. This gives a kernel process the capability to more precisely control its flow by only allowing certain communications to occur.

To provide timely response to events, the process receiving the event is interrupted and the state of its activities is saved by the kernel. After

the event is acted on, the process is restored to its original state and continues from the point where it was interrupted.

2.2 Messages

Messages are the primary method of communication between cooperating processes in the operating system. The DMERT kernel provides a variety of OSTs for allocating, sending, receiving, canceling, auditing, and freeing messages for individual processes.

A message consists of a fixed-size header followed by a variable-size block of data, called the message body. The header contains routing directions, status indicators, and other administrative information. The message body's contents and structure are determined by the cooperating processes using the message.

A reserved event is used to notify a process that it has received a message. Hence, when a process receives a message, control is passed to its event entry. Waiting messages are queued in a message buffer segment in the kernel's address space and are directly accessible by kernel processes. However, kernel processes do not manipulate messages directly without first dequeueing them by using one of the message dequeueing OSTs. Supervisor processes do not have direct access to the message buffer segment. Messages must be copied into and out of this segment from and to the supervisor's own space.

2.3 Ports

Communications by messages require the knowledge of the target process's Process Identifier (PID). Since processes can be created and terminated dynamically, PIDs also are generated dynamically and a process typically knows only its own PID. Process ports permit processes to communicate with each other without knowing each other's PID. A process port is a globally known "device" to which a process may attach itself for receiving messages. Other processes may communicate with a process connected to a port by sending a message to that port rather than to a specific PID. Thus, process ports permit unrelated processes to communicate with each other.

2.4 Faults

Faults are very similar to events. A fault is a one-bit piece of information that may be sent, received, and recognized by a process. DMERT has 32 fault bits, some of which are reserved for special meanings by DMERT. When a process's fault bits are set, control is passed to its fault entry.

Faults are used to inform processes of error and overload conditions and take precedence over events. If a process has both events and faults pending, control will be passed to its fault entry first.

2.5 Interprocess traps

As mentioned in Ref. 1, an OST is generally used to transfer control to the kernel to perform some service on behalf of the requesting process. As such, an OST does not constitute a form of interprocess communication. An interprocess trap is a generalized notion of the OST, where a process can trap to some other process rather than the kernel to have some service performed. Because this interprocess trap can communicate service requests or other data between two processes, it is a form of IPC.

Interprocess traps are more restrictive than other IPC mechanisms. A process can trap to another process only if the former process's execution level is lower than the latter's. Control is passed to the trapped process's OST entry.

2.6 FIFOs

Unrelated processes can use FIFOs for character stream communication. Conceptually, a FIFO is a queue of characters; characters flow through in a first in-first out order (hence, the name). The queue is fixed in length and may become filled if writers write faster than readers read. When such a condition occurs, further writes are prevented. Similarly, reads are prevented when the queue is empty.

Each FIFO has a name; that is, each is a special file in the file system (described in Section VIII). This naming characteristic distinguishes it from a pipe,[2] because only processes spawned by a common ancestor can communicate using a pipe. Naming allows any process that knows a FIFO's name to use it, regardless of its ancestry.

Any process level may use this mechanism. The FIFO is a suitable replacement for messages where communication is a character stream.

2.7 Shared memory

A segment is the basic unit of shared memory. It can be shared between different processes, or between multiple instances of the same process. A segment also can be shared among any number of arbitrary processes by assigning a global name to it and allowing any process to access it that has been given the global name.

Since it is completely managed by the cooperating processes, shared memory is the form of interprocess communication most susceptible to error, but it is also the most efficient. Within the operating system, shared segments are used wherever reliability will not be sacrificed and where real-time response is paramount.

III. MEMORY MANAGEMENT

DMERT memory management centers around the segment. A segment is a set of logically related pages. A page is 2048 bytes of

contiguous main memory that always begins on addresses that are multiples of 2048. A segment is a collection of pages that do not have to be physically contiguous. All segments in the system are unique, that is, no page can belong to more than one segment. (Reference 3 contains a detailed description of pages, segments, and memory translation.)

3.1 Segment attributes

Segments are created in main memory by the operating system on demand and disappear when they are no longer needed. When the initial image of a process is created, its segments are loaded into main memory by the memory manager. Although a segment of a user or supervisor process can be swapped out to a swap area reserved on the disk if additional main memory is needed for a process of higher priority, the segment remains known to the system until it is not required by any process.

A segment is identified internally by its Segment Identifier (SID), which is the virtual address of its entry in the kernel's segment description table. A segment of a supervisor process also can be referenced by a segment number that indexes into the process's segment list. The segment list specifies the segments that are known to the process, including those that are not necessarily in the current virtual address space of the process.

A process has segments that may be private or shared by other processes. Text segments are shared when possible to avoid duplication of commonly used functions. Data segments are usually private since most processes want sole access to their own data. However, a data segment can be shared with another process for implementing interprocess communication (see Section 2.7) and can have a global name. Segment sharing increases the efficiency of memory management since a process needing to use a shareable segment already in main memory need only attach the segment to its virtual address space, thus reducing segment swapping.

A segment can be defined as executable, readable, writable, or some combination thereof. Two different processes sharing a segment can have different access rights to it. So, for example, an "owner" of the segment could have read and write access, while another "user" of the segment would only be allowed to read it.

A segment also can be designated as swappable or nonswappable. If nonswappable, the segment will not be swapped from the main memory to disk memory. If it is imperative that the segment not be moved around in main memory as, for example, when I/O is being done in the segment, it can be locked in main memory. In addition, a segment can be blocked (that is, made unavailable) to other processes while it is being initialized.

Finally, segments can be either active or inactive. If inactive, the segment is not presently a part of the address space for each process that owns it.

3.2 Process images

The executable image of a process consists of a set of segments. Originally, a kernel process had its segments completely specified at link time. Later enhancements allow kernel processes to dynamically add or drop segments using messages to the memory manager. A supervisor process can use OST calls to create or destroy segments, make copies of segments, request named segments, attach or detach segments to its address space, grow or shrink a segment, lock a segment in main memory for I/O, or make a segment swappable or nonswappable.

A kernel process starts with at least four segments: the Kernel Process Control Block (KPCB) segment, the stack segment, the system message buffer segment, and the process's text segment. The KPCB segment contains the process-dependent information. The stack and system message buffer segments are shared with all other kernel processes and the DMERT kernel while the process's text segment is shared only with other invocations of the same kernel process. (Most kernel processes have only one invocation active at a time.)

A supervisor process starts with at least three segments: the Process Control Block (PCB) segment, which contains the system's tables for the process; the stack segment; and the text segment. The process's data may be combined with its stack into a single segment or it may be placed in a separate segment. The access modes for the PCB and text segment are read-only and read-execute, respectively. The process may never write its PCB. The stack segment of a supervisor process is never shared, while a text segment is usually shareable between multiple invocations of the same process.

IV. SCHEDULING

DMERT simultaneously supports both a real-time and a time-sharing environment. The kernel and kernel processes operate in a real-time environment and have first call on the available time of the 3B20D processor. The remaining time is shared among special, supervisor, and user processes.

4.1 Real time

Any process that must satisfy a real-time requirement should be a kernel process. DMERT maintains a process hierarchy based on 16 execution levels. A kernel process can belong to levels 3 through 15. (Levels 0 through 2 are reserved for time-sharing environment.)

DMERT bases its real-time allocation strategy on three concepts: execution levels, round robin scheduling, and preemption. DMERT dispatches processes at the highest execution level first. For example, a process belonging to execution level 15 is dispatched before a process belonging to level 14, which is dispatched before a process at level 13, and so on. For each execution level, DMERT maintains a list of waiting processes. When a process requests servicing, it is added to the appropriate list in a round robin fashion. As the operating system descends the hierarchy, it dispatches all the waiting processes at each level.

Generally, a kernel process executes until it exits. However, if another kernel process at a higher execution level is awakened, DMERT preempts the executing process. Upon completion of the preempting process, if no other higher level processes are awakened, the operating system resumes the suspended process.

DMERT's management of real time is straightforward and adds only a minimal amount of overhead. In fact, preempting one kernel process and dispatching another takes only about 320 microseconds. At the same time, applications are allowed to assign their own process's execution levels, which customizes their control and distribution of real time. This approach has proved to be quite flexible and permits a variety of applications.

4.2 Time sharing

As stated previously, the portion of real time not utilized by the kernel or kernel processes is time shared among supervisor and user processes. Processes supporting the time-sharing environment, such as the scheduler and the memory manager, are special kernel processes that reside at execution level 2. These processes are at the bottom of the real-time hierarchy and gain control of the processor only after all other real-time work is completed.

Supervisor and user processes normally execute at level 0. Level 1 is reserved for when they need to lock each other out. Within execution level 0, the scheduling hierarchy of supervisor and user processes is based on priority. The major difference between priority in the time-sharing environment and execution levels in the real-time environment is that DMERT adjusts priorities dynamically, whereas execution levels are fixed.

Priority adjustment is based on two factors: state and age. A time-sharing process can be in one of three states: sleeping, waiting, or executing. As a process enters a new state, it begins to age (from zero) until it changes state. Processes in different states age at different rates.

DMERT uses age and state to adjust the process's scheduling

priority. In particular, the priority of a sleeping or executing process is reduced as it ages and the priority of a waiting process is increased as it ages. The result is that a low-priority process waiting to run will eventually move above higher priority processes that have been executing.

DMERT selects the time-sharing process to execute by searching down a priority list until it finds a process waiting to run. The selected process continues to run until it changes state or is preempted by a kernel process. It cannot be preempted by a higher priority time-sharing process. A time-sharing process can leave the executing state for one of the following reasons: (*i*) it roadblocks (enters the sleeping state); or (*ii*) it uses up its allotted time slice and is returned to the waiting state; or (*iii*) it terminates. When a process times out, it is chained to the end of the list of waiting processes at the appropriate priority. Then, DMERT searches the priority list to find the next time-sharing process that is waiting to execute.

V. PROCESS CREATION AND TERMINATION

The DMERT operating system supports the dynamic creation and termination of kernel, supervisor, and user processes. A process can be created on demand from a process load file maintained in secondary memory and can be terminated and recreated at will. Process creating and termination can be requested via messages, OSTs, or commands executed at a terminal.

5.1 Process creation

The load file of a process is generated by means of the process loader, which is part of the software development system.[4] A process load file contains the text and data segments that comprise the initial image of the process. It also specifies a variety of basic process attributes, such as the execution level, stack size, instruction privileges, entry points, and, for time-shared scheduling, priority and time slice.

The creation of a process involves the identification of each of its segments in the kernel memory management tables, the formation of its initial image in main memory, and the identification of the process itself in the operating system scheduling tables. Start-up of a created process is generally accomplished by sending it a special initialization event, in which case control is passed to the process's event entry. In the case of supervisor and user processes, control is passed to the process's start entry.

5.1.1 Process creation mechanisms

There are three distinct mechanisms for creating a process. The system bootstrap mechanism creates a set of processes that constitute

the nucleus of the operating system and are essential to its operation. These include a process, called the process manager, whose principal function is the creation of other processes from process load files. The process manager can be instructed to create processes via the mechanism, and of sending it a message containing the name of the process load file and other relevant information. The third mechanism is the execution of a "fork" system call by a user process that results in the creation of a copy of the process executing the fork. The copy, called a child process, can subsequently execute another file via the "exec" system call, i.e., exchange its text and data for that of another process load file.

The creation of processes during system bootstrap is carried out by the kernel in a very efficient manner utilizing full access to the memory management tables and functions. The process manager is implemented as a swappable supervisor process that calls upon the services of the memory manager through OSTs and the file manager through messages. The fork mechanism is carried out within the process executing the fork and avoids the overhead associated with loading in the process manager.

5.1.2 Process creation features

Certain features are provided by the process creation mechanism. These include a distinction between single-copy and multiple-copy processes, the sharing of segments, and the use of shared libraries.

Single-copy processes are not replicated in main memory by successive creations. Instead, the operating system increments a usage count, which is decremented whenever the process is terminated. The process remains in the system so long as its usage count is at least 1. Device handlers that are repeatedly opened and closed are typically single-copy processes.

Successive creation requests for a multiple-copy process results in processes with distinct PIDs that have independent existences. As mentioned earlier, the text segment is usually shared among the invocations of a multiple-copy process. Data segments are typically private.

Processes that have a parent-child relationship may conveniently share segments as part of the creation process. The process load files of the parent and child processes define the segments that are shareable. When the parent process requests the creation of the child via a message to the process manager, it identifies the segments to be shared and specifies the access permissions for the child process. A process can share segments with any number of different child processes.

Shared libraries consist of segments of text and data that can be incorporated into any number of processes. A shared library is formed

by the process loader, and its process load file is maintained in secondary memory. If a process wants to use a shared library, the load file of the process need only contain the name of the library. The segments of the library are loaded into the process when the process is created unless they have already been loaded to satisfy the creation of another process. Symbolic references are resolved when the process load file is formed. The use of shared libraries decreases the size of the text segments in the process load file.

5.2 Process termination

Any process running under DMERT may elect to terminate itself or may be terminated by request of another process or the kernel. Exceptional cases are allowed; that is, the operating system has the capability of declaring processes to be essential or nonterminable. In particular, the processes that provide the basic operating system services are essential and cannot be terminated short of another system initialization. A kernel or supervisor process may be declared to be nonterminable as an option of the process loader. For such processes termination requests are always denied.

The operating system supports a variety of mechanisms for requesting process termination, as described below. However, regardless of the specific request mechanism used, the operating system takes the same basic actions.

5.2.1 Termination actions

All requests to terminate a process eventually result in a message being sent to the system scheduler specifying the PID of the process to be terminated. The termination request will result in the removal of the process from the system, unless it is a single-copy process with a usage count greater than 1, in which case the usage count is decremented. Removal of the process entails deleting its entry in the scheduling and segment tables. Shared segments that are needed by other processes for execution are retained in the system. The other segments are removed from the memory management tables and any associated secondary memory uses for swapping are freed up.

The operating system also takes some actions on behalf of the terminating process to free up other system resources that may have been allocated to it. Messages still queued up for the process are deleted or returned to the sender with a special acknowledgment. The process is detached from system ports and interrupts. For a supervisor process, the operating system issues close requests for any files it may have in the open state.

The message sent to the scheduler to terminate a process may optionally request a memory dump of the process. In this case, the

operating system produces a file in secondary memory similar to a process load file, which includes a copy of each of the process's segments at the time of termination. Such dumps are often useful for debugging purposes. Another option frequently exercised to coordinate the interactions of processes is to have the parent process notified when one of its child processes is terminated.

5.2.2. Process termination request mechanisms

A process may send a message to the system scheduler to terminate itself or another process. In certain situations, the operating system will terminate a process automatically. For example, a process that is faulted but that does not have a fault entry is terminated by the kernel.

A user process typically executes an exit system call to terminate itself. The exit system call is converted into a terminate message to the scheduler.

The operating system provides various OSTs to terminate a group of processes. One OST can be used by supervisor or kernel processes to terminate all processes having the same Utility Identifier (UID). (The UID is an identifier of a process that is selected at process load time and is administered to be unique to each process load file). This OST terminates all invocations of a multiple-copy process, since multiple invocations have the same UID. Another OST can be used by a kernel process to terminate all processes belonging to a specified process class. DMERT supports the optional grouping of processes into one of several classes, with each class chosen at load time. The capability of terminating all processes in a given class is useful in overload control schemes. Another termination OST available to kernel processes is specifically designed to handle system message buffer overload. This OST terminates processes that are using more than a specified percentage of the system message buffer resources. The actions of this OST also can be restricted to processes with a specified execution level or process class.

Several commands are provided to permit process termination from a terminal. One command will force the termination of any process with a specified PID and also will cause a memory dump of the process to be created. Another command is provided to terminate all processes with a specified UID. It also has the special feature of clearing the names of all shared segments used by the processes. This feature is used in conjunction with the updating of a process.[5]

VI. SIMULATED *UNIX* OPERATING SYSTEM

The time-sharing environment supported by DMERT simulates a *UNIX* operating system[2] implemented via a supervisor process called the *UNIX* Supervisor Process (USP). The USP supports standard

UNIX software including system calls from C programs, file operations, process communication through pipes, and interpretation of terminal commands through the "shell" process.

Processes controlled by the USP are called user processes. The USP partitions its address space into a user area and a supervisor area and appears as a single process to the DMERT kernel. Thus, the user and supervisor are physically combined into the same process, having the same PID, scheduling priority, PCB, etc. Each time a user process forks, another USP is formed.

The role of the USP is to supply services to its user portion. It accomplishes this through supervisor OST calls and through communication with other DMERT processes. For example, file system capabilities are provided by the USP sending the appropriate messages to the DMERT file manager process.

The availability of a simulated *UNIX* operating system in DMERT allows *UNIX* programs from other processors to execute on the 3B20D Processor. DMERT provides some capabilities to user processes not currently supported by the standard *UNIX* operating system. These include asynchronous I/O directly to or from the user's address space and memory management of user process segments. In addition, there are a number of file system capabilities, such as contiguous files, that are provided through the DMERT file management facilities discussed in Section VIII. User processes also have access to the DMERT IPCs such as messages and events.

DMERT's memory management capabilities allow a user process to manipulate and share portions of its address space on a segment basis. In particular, a user process can create a new segment in its address space and can specify the virtual address of the segment. It can acquire an existing and named segment into its address space and also remove segments from its address space. Segments listed in a user process's PCB can be activated or deactivated through OSTs to the USP. OSTs permit it to share up to three segments with a process it creates via the DMERT process creation functions described earlier.

VII. I/O FACILITIES

The operating system supports communication with peripheral devices through a set of drivers and device handlers. These drivers isolate most processes from the details of the peripheral system, and they ensure efficient use of the peripheral devices by scheduling access to them on an equitable basis.

The architecture of the I/O software closely resembles the I/O hardware architecture.[5] The I/O Processor (IOP) driver and the device handlers manage the IOPs, the Peripheral Controllers (PC), and the

Peripheral Controller Subdevices (PCSDs). The disk driver manages and controls the disk file controllers and the disks.

7.1 Input/output processor driver

The IOP driver is a kernel process that administers all IOP transactions in the 3B20D DMERT system. The driver is responsible for normal I/O activities fault recognition and recovery, configuration management, and diagnostic access.

The IOP, from a software standpoint, can be visualized as a three-level structure (see Fig. 1). The "front-end" Peripheral Interface Controller (PIC) controls up to sixteen peripheral controllers (PCs), and

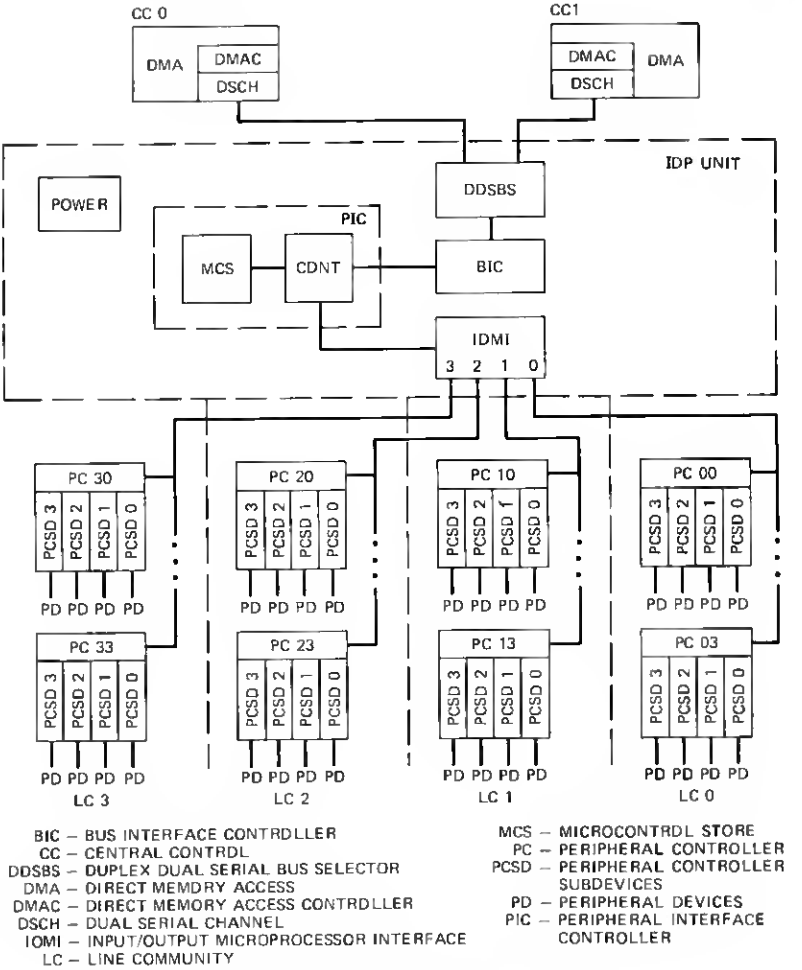

Fig. 1—Input/output processor.

each PC controls up to four subdevices (PCSDs). The subdevice provides the interface to the end device, such as magnetic tape unit, teletypewriter (TTY), data link, etc. Four PCs are combined to form a PC "community" with each PC community having a separate power supply.

Each element in the IOP (PIC, PC, and PCSD) has a corresponding element in the driver called a handler and corresponding unit control and option blocks in the Equipment Configuration Data (ECD) data base.[6] The handler for the PCSD is referred to as the device handler. A handler for a PC is called the generic PC handler or application PC handler. The handler for the PIC is called the generic PIC handler. The term generic implies that the handler is capable of performing all required handler functions for more than one PC type (that is, TTY, magnetic tape, data link, etc.).

Handlers are collections of C-language functions that have well-defined interfaces with the driver and are responsible for carrying out all maintenance (excluding diagnostics), recovery, and normal mode operations for their respective elements in the IOP. Handlers contain the necessary specialized logic to deal with a given unit type.

The handlers' service routines, input routines, control routines, and associated libraries form a single IOP driver process (IODRV).

IODRV can be subdivided into the following functional areas:

(*i*) Common service routines: routines that are frequently called from numerous points within IODRV and the handlers.

(*ii*) Configuration control: routines that maintain proper configuration of IOP units (inverted tree structure).

(*iii*) Input routines: routines that process primary inputs from the DMERT operating system and pass them off to IODRV configuration control or handlers.

(*iv*) Application and generic handlers: the operational interface between the user and physical device (magnetic tape, terminals, etc.).

(*v*) Maintenance handler: the diagnostic interface to the IOP units.

(*vi*) Archive libraries: system routines used by IODRV and other kernel-level processes.

Normal mode activities are carried out through the input routines, common service routines, and the operational handlers. All communications within the driver are through function calls.

I/O messages enter IODRV as message events and pass through the message input routines. Similarly, operating system traps enter IODRV at the OST entry point and pass through the OST input routines. Typically, I/O messages and OSTs contain a Logical Device Identification Number (LDIN) that identifies a logical (or virtual) device with which a user wishes to do I/O. IODRV maps the LDIN to

one or more physical devices. Once a physical device is identified, the IODRV can identify the corresponding handler via the ECD and pass control to it.

Completion reports, or responses, are deposited in the IODRV response queue by the IOP. If responses have been added to the response queue within a certain batching interval, the IOP will interrupt the 3B20D, causing IODRV to be entered. IODRV pops each queued response and, based on the PC and PCSD identifier in the response, maps to a physical unit, and passes control to the handler.

Maintenance and recovery activities are coordinated through IODRV configuration control routines, which, in turn, call on the handlers at appropriate points in time to carry out specialized maintenance operations at the subdevice level. Diagnostics for the PIC and PC are handled exclusively by the maintenance handler.

7.1.1 Handler applications

Peripheral devices supported by the IOP include TTY terminals, Maintenance TTY (MTTY) terminals, magnetic tape drives, data links, and the Scanner and Signal Distributor (SCSD). Interfaces to these devices are provided by IODRV and device handlers. IODRV is responsible for initializing units upon bootstrap and removing and restoring units upon manual requests or faults. Each handler-peripheral device combination determines the interface mechanism and the set of features to be supported. The following sections give a brief description of the facilities supported by each peripheral device type.

7.1.2 Terminal devices

The Craft Interface Handler (CIH) provides access to terminal devices. The CIH communicates with two types of controllers: Maintenance Terminal Controllers (MTTYCs) and Terminal Controllers (TTYCs). MTTYCs support four subdevices: a Maintenance Terminal (MTTY); a Receive-Only Printer (ROP); a Switching Control Center (SCC) interface; and an Emergency Action Interface (EAI).[7] TTYCs support terminals (TTY).

The MTTY, TTY, and ROP are known as terminal devices. The SCC and EAI devices are not terminal devices and are accessed via other handlers (see below). All standard terminal operations supported by the *UNIX* operating system are available to TTY devices, including read, write, open, and close requests, which are supported through a message interface. The ROP does not support reads.

7.1.3 Data links

The Communication Protocol Handler (CPH) provides access to synchronous data links. The CPH is designed to communicate with

two types of peripheral controllers: (*i*) the MTTY controller, and (*ii*) the synchronous data-link controller. In the case of the MTTY, access is available only to the SCC peripheral controller subdevice, which supports synchronous data link communication between the 3B20D and an SCC office using the BX.25 protocol.

The synchronous data link controller supports the BX.25 link layer (level 2) communication protocol and the Digital Data Communication Message Protocol (DDCMP) through the use of different versions of peripheral controller software. The CPH software supports two access methods: link-layer protocol access (level-two-only access) and BX.25 packet-level (level 3) protocol access. The use of a link-layer protocol (BX.25 and DDCMP) assures the integrity of data transmissions on a physical link. The use of a packet-layer protocol (BX.25) allows the added capability of multiplexing multiple users on a physical link. Flow control procedures also are used on both protocol layers.

In addition to the the two different access methods, the handler supports both a simplex and a duplex link configuration. In the duplex configuration, two physical links make up a logical communication path between the 3B20D and another system. The CPH automatically routes data through the currently active physical link. Link switching is done automatically when the active link fails.

7.1.4 Magnetic tape drives

The magnetic tape peripheral controller handles up to four 9-track 800 or 1600 bits per inch (bpi) tape drives. The magnetic tape handler provides the interface to this controller and supports open, read, write, seek, and close requests through a message interface. Seeks are not supported for write operations.

7.1.5 Scanner and signal distributor

Administration and control of the Scan and Signal Distributor (SCSD) points currently involves two DMERT kernel processes: the SCSD administrator and the SCSD handler. The latter is an integral part of the I/O driver process. The primary function of the SCSD software is to provide an interface enabling client processes to manipulate distribution points and receive information about the state of the scan points (i.e., autonomous scan state transition and directed scan reports). The SCSD administrator allows a client process to identify SCSD points by logical or physical addresses; logical addressing allows applications to code software independently of physical cabling.

The SCSD handler translates messages from the administrator into SCSD controller commands and receives responses from the controller

and forwards these responses to the administrator through a message interface.

7.1.6 Direct user interface

For some applications the current method of communication with the peripheral controller subdevices through IODRV is not efficient enough to meet their needs. Therefore, the Direct User Interface (DUI) exists to expedite data transfers between an application process and a specialized 56-KB BX.25 data-link controller.

The DUI handler is an integral part of IODRV. In the normal mode of operation, the only functions of the DUI handler are to set up and clean up the DUI table, which is in a common area of memory and is used for passing commands and status information between the application process and the peripheral controller. Using the DUI table, jobs are passed directly to the peripheral controller by the application process without any intervention from IODRV.

A secondary function of the handler is to administer the fault recovery strategy for the peripheral controller subdevice. If the subdevice has to be removed or restarted, the handler will tear down the DUI table and send a message to the application process.

7.2 Disk driver

The disk driver is a kernel process that handles all normal disk I/O and all maintenance disk I/O. Only system initialization I/O bypasses the disk driver and transfers information directly from the system boot device to main memory. The disk subsystem consists of the disk driver, the Disk File Controller (DFC), and the Moving Head Disk (MHD) drives. The 3B20D supports a maximum of eight DFCs, each having up to eight MHDs. The DFC and MHDs are described in Ref. 8.

MHDs may be used in a simplexed or duplexed configuration. In simplex mode the MHD stands alone. Should a file hecome damaged it will be irretrievably lost. In duplex mode two MHDs are maintained such that each is an exact copy of the other. Should one disk fail the other can be used in simplex mode.

7.2.1 Operational characteristics

The disk driver handles open, close, read, and write message requests. Open and close messages are passed from the file manager (see Section VII) to the disk driver, while read and write requests may be sent directly from any process or routed through the file manager. Before the kernel attaches the read or write message to the disk driver's message queue, it verifies that the segment is locked in main

memory (see Section 3.1). It also verifies that the I/O transfer is within the bounds of the segment.

When the disk driver processes the I/O message, it translates the LDIN contained in the message to one or more physical devices. The request is then placed in one of three circular job submit queues in main store associated with the DFC for each specified physical device.

The three types of disk job queues are high-priority, base-priority, and special. Special commands sent by maintenance processes or originated in the disk driver are immediately executed by the DFC from the special job queue. High-priority jobs can be sent by any client process and will be processed by the DFC before base-priority jobs. All other jobs are placed in the base-priority queue.

Whenever the DFC completes a job requested by the driver, it returns a response indicating the outcome of the job. All job responses are placed in a single main store response queue, regardless of the priority of the original job. The DFC generates an interrupt to the driver after each response is added to the queue. The driver only clears the interrupt after processing the last entry in the response queue.

The disk driver handles job responses each time it is entered at its interrupt entry. The job response indicates the status and identity of the job being reported. If the job was successful, the driver sends a successful job completion acknowledgment to the client using the same message buffer that requested the I/O. In writing to duplexed disks, the driver guarantees that both disks were written successfully before acknowledging the job. In reading from duplexed disks, the driver reads from a single disk, alternating disks between requests.

If a job failed, the driver determines whether the device should be removed from service or if it should retry the job.

When the driver wishes to retry a failed job, it sets up a retry request in the main store retry queue. The format of an entry in the retry queue is the same as that of a job in any of the other queues. After writing the entry in the queue the driver wakes up the DFC with a programmed I/O command. The DFC then takes this job, even if the other submit queues contain work. When the driver handles the response from the retry request, it knows the queue is available for reuse.

7.2.2 Reliability characteristics

The disk driver also has a message interface for maintenance commands. Once the device (MHD or DFC) is taken out of service by the disk driver, the device can be reserved for maintenance access and the disk driver provides the maintenance client processes unlimited access to the device. During maintenance, specific areas of a MHD can be read or written by bypassing many of the operational checks performed

on normal I/O requests. This allows the creation of a disk and the system update of a disk with a new software generic.[6]

VIII. FILE SYSTEM

All accesses to the file system are done through the file manager, a DMERT kernel process. In addition to maintaining file system security and integrity, the file manager translates read and write requests within the file system to physical I/O requests on the disk.

The DMERT file system is similar to the file system provided by the *UNIX* operating system and features a hierarchical structure, byte-oriented files, and uniform access to files, directories, and periphery. In addition to regular files, which are scattered throughout the disk and can grow dynamically, DMERT also provides contiguous and extent files, which are contiguous on disk but have limits on their growth. Contiguous and extent files are optimum for data base and object files, where large, fast I/O transfers are needed. For field update, DMERT provides a "windowless move" facility, which automatically moves an updated object file over the old one, thus eliminating any possibility that the file be used or the system initialized while the file is in an inconsistent state.

To meet DMERT's reliability requirements, DMERT file systems are crash resistant. In particular, a crash does not jeopardize file system integrity, the file systems do not need manual repair, and they are available within seconds after a crash.

The file manager uses two techniques to ensure crash resistance. First, it orders all writes to disk to maintain a consistent file system state. To create, link, or write a file, the ordering is:

(*i*) Write the data blocks
(*ii*) Write the indirect i-node blocks
(*iii*) Write the i-node*
(*iv*) Write the directory entry, if necessary.

To unlink or truncate a file, the ordering is:

(*i*) Write the cleared directory entry, if necessary.
(*ii*) Write the cleared i-node.
(*iii*) Free the blocks.

Second, to ensure that no block is allocated to more than one file, the file manager rebuilds a file system's free-block list before it is used following a crash. Doing this for the 50-000 block, 2048-i-node *root* file system adds about 10 seconds to DMERT's boot procedure.

These two techniques are sufficient to ensure crash resistance, and we have found no problems with these in the field.

---

* An i-node describes a file and contains its block addresses. An indirect i-node block extends the i-node and contains more block addresses.

IX. SUMMARY

This article has described the DMERT nucleus, which consists of the kernel, the special processes, the I/O drivers and file manager, the process manager, and the *UNIX* supervisor. The major services provided by this nucleus include a multitude of interprocess communication mechanisms, a sophisticated set of memory allocation features, both real-time and time-shared scheduling, dynamic process creation and termination, a simulated *UNIX* environment's communication with terminals, magnetic tape drives, data links and disks, and powerful real-time and time-shared file system capabilities. The operating system has been continually evolving since DMERT was conceived, and is expected to continue to evolve over the next few years. This article has described the first official version of DMERT, which entered service in the Bell System during September 1981.

X. ACKNOWLEDGMENTS

We thank C. J. Antonelli, J. Q. Arnold, T. P. Bishop, R. W. Fish, N. A. Martellotto, R. R. Snead, P. J. Stankus, R. M. Venzon, and R. E. Yuknavech for their contributions to this document. Special thanks go to J. J. Wallace for his contributions to the file manager section.

REFERENCES

1. J. R. Kane, R. E. Anderson, and P. S. McCabe, "The 3B20D Processor & DMERT Operating System: Overview, Architecture, and Performance of DMERT," B.S.T.J., this issue.
2. D. Ritchie and K. Thompson, "The *UNIX* Time-Sharing System," B.S.T.J., *57*, No. 6, Part 2 (July–August 1978), pp. 1905–29.
3. I. K. Hetherington and P. Kusulas, "The 3B20D Processor & DMERT Operating System: 3B20D Memory Systems," B.S.T.J., this issue.
4. B. R. Rowland and R. J. Welsch, "The 3B20D Processor & DMERT Operating System: Software Development System," B.S.T.J., this issue.
5. A. H. Budlong and F. W. Wendland, "The 3B20D Processor & DMERT Operating System: 3B20D Input/Output System," B.S.T.J., this issue.
6. R. H. Yacobellis, J. H. Miller, B. G. Niedfeldt, and S. S. Weber, "The 3B20D Processor & DMERT Operating System: Field Administration Subsystems," B.S.T.J., this issue.
7. M. E. Barton and D. A. Schmitt, "The 3B20D Processor & DMERT Operating System: Craft Interface," B.S.T.J., this issue.
8. R. E. Haglund and L. D. Peterson, "The 3B20D Processor & DMERT Operating System: 3B20D File Memory Systems," B.S.T.J., this issue.